

Lernzettel

Interprozesskommunikation und IPC-Mechanismen: Pipes, Sockets, Shared Memory, Message Queues

Universität: Technische Universität Berlin
Kurs/Modul: Systemprogrammierung
Erstellungsdatum: September 6, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Systemprogrammierung

Lernzettel: Interprozesskommunikation und IPC-Mechanismen

(1) Überblick und Ziele. Interprozesskommunikation (IPC) dient dem Austausch von Daten und der Synchronisation zwischen Prozessen auf demselben Rechner. Typische IPC-Mechanismen sind Pipes, Sockets, Shared Memory und Message Queues. Die Wahl des Mechanismus hängt ab von Latenz, Durchsatz, Kopieraufwand, Synchronisationsbedarf und Sicherheitsanforderungen.

(2) Pipes. Pipes ermöglichen eine unidirektionale Datenleitung zwischen zwei Prozessen. Typischerweise entsteht eine Pipe durch `fork()` und wird von Eltern- und Kindprozess genutzt.

Anwendungsarten

- Anonyme Pipes: `pipe(fd[2]);` Enden `fd[0]` (Lesen) und `fd[1]` (Schreiben).
- Named Pipes (FIFO): `mkfifo(name, mode);` existieren im Dateisystem und können von unabhängigen Prozessen geöffnet werden.

Eigenschaften

- Streams orientiert (keine Nachrichtenstruktur erzwingen).
- Blockierend oder nicht-blockierend je nach Modus.

```
int fd[2];
pipe(fd); // fd[0] = Leseende, fd[1] = Schreibeende
if (fork() == 0) {
    close(fd[0]);
    write(fd[1], "Daten", 5);
} else {
    close(fd[1]);
    char buf[6];
    read(fd[0], buf, 5);
}
```

Beispieleinsatz Kurzlebige Kommunikation zwischen Parent/Child, Pipelining von Prozessen.

(3) Sockets. Sockets ermöglichen IPC auf demselben Rechner (Unix-Domain) oder über Netzwerke.

Typen

• *AF_UNIX* (lokal, domneninterner IPC) Typen von Verbindungen

- `SOCK_STREAM` (orientiert, zuverlässig)
- `SOCK_DGRAM` (datenpaketorientiert, unzuverlässig)

Grundlegender Ablauf

- Socket erstellen: `socket(family, type, protocol)`
- Server-Seite: `bind()`, `listen()`, `accept()`

- Client-Seite: `connect()`
- Daten senden/empfangen: `send()/recv()` oder `write()/read()`

Nebenbemerkung Für lokale IPC oft Unix-Domain Sockets verwenden; für verteilte Systeme TCP/UDP einsetzen; zur Sicherheit ggf. Verschlüsselung (TLS).

```
int s = socket(AF_UNIX, SOCK_STREAM, 0);
struct sockaddr_un addr;
memset(&addr, 0, sizeof(addr));
addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, "/tmp/ipc.sock", sizeof(addr.sun_path)-1);
bind(s, (struct sockaddr*)&addr, sizeof(addr));
listen(s, 5);
int client = accept(s, NULL, NULL);
send(client, "Hi", 2, 0);
char buf[3];
recv(client, buf, 2, 0);
```

(4) Shared Memory. Gemeinsam genutzter Speicherumfang, der direkte Zugriff mehrerer Prozesse ermöglicht.

POSIX vs. System V

- POSIX: `shm_open`, `ftruncate`, `mmap`, `shm_unlink`
- System V: `shmget`, `shmat`, `shmdt`, `shmctl`

Wichtig Shared Memory bietet hohen Durchsatz (wenige Kopien) erfordert jedoch explizite Synchronisation (z. B. Semaphoren, Mutex im gemeinsamen Speicher).

```
int fd = shm_open("/myshm", O_CREAT|O_RDWR, 0666);
ftruncate(fd, 4096);
void *p = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
...
munmap(p, 4096);
shm_unlink("/myshm");
```

(5) Message Queues. Message Queues ermöglichen das asynchrone Versenden strukturierter Nachrichten zwischen Prozessen.

POSIX vs. System V

- POSIX: `mq_open`, `mq_send`, `mq_receive`, `mq_close`
- System V: `msgsnd`, `msgrcv`, `msgget`, `msgctl`

Eigenschaften

- Nachrichten haben Typ/Type-Feld, feste Größen oder variabel mit Limits.

- Oft asynchron, sortierte Warteschlangen, Berechtigungsvergabe (permissions).

```
mqd_t mq = mq_open("/myqueue", O_CREAT|O_RDWR, 0666, NULL);
mq_send(mq, "MSG", 3, 0);
char buf[4];
mq_receive(mq, buf, 3, &prio);
mq_close(mq);
mq_unlink("/myqueue");
```

(6) Vergleich und Design-Empfehlungen.

- Kopien vs. kein Kopieren: Shared Memory minimiert Kopien, erfordert Synchronisation.
- Latenz vs. Datenvolumen: Pipes/Sockets gut für Streaming; Message Queues für strukturiertes Messaging.
- Komplexität der Synchronisation: Shared Memory benötigt robuste Synchronisation (Semaphoren, Mutex, Barrieren).
- Sicherheit: Zugriffsrechte (Permissions), ggf. Verschlüsselung bei Netzverkehr.

(7) Musteranwendungen (Beispiele).

- Producer-Consumer mit Shared Memory + Semaphoren
- Pipe-basierte CLI-Kommandokette (Pipes in der Shell)
- Client-Server-Kommunikation über Unix-Domain Sockets

(8) Sicherheit und Betrieb in der Praxis.

- Zugriffskontrollen über Berechtigungen (fs-Module, IPC-Objekte)
- Minimalrechte und sauberer Ressourcen-Release (unlink/shm_unlink, close, waitpid)
- Robustheit gegen Partial Failures und Deadlocks durch Timeouts und Timeout-Handling